FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH-LEVEL SYNTHESIS

by

Firas Abdul Ghani

Submitted to the Graduate School of Engineering and Natural Sciences in partial fulfillment of the requirements for the degree of Master of Sciences

Sabancı University
August 2017
FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS
USING VIVADO HIGH LEVEL SYNTHESIS

APPROVED BY:

Assoc. Prof. Dr. İker Hamzaoğlu
(Thesis Supervisor)

Assist. Prof. Dr. Murat Kaya Yaşıcı

Assist. Prof. Dr. Baykal Sarıoğlu

DATE OF APPROVAL: 01/08/2017
To my Mother and Father
ACKNOWLEDGEMENT

Foremost, I would like to express my sincere thanks to my advisor Dr. İlker Hamzaoğlu for his endless support during my MS study. I respect very much his suggestions and comments that improved my design skills more and more. His motivation, and immense knowledge enabled me to finish my thesis with very good results.

In addition, I would like to really thank Ercan Kalalı for his endless support and advices during my MS study. I also want to thank the other members of System-on-Chip Design and Testing Lab, Ahmet Can Mert, and Hasan Azgün for their support during my MS study.

I also want to thank my family and my wife who encouraged me a lot during my thesis.

Finally, I would like to acknowledge Sabancı University and Scientific and Technological Research Council of Turkey (TUBITAK) for supporting me with scholarships throughout my studies. This thesis was supported by TUBITAK under the contract 115E290.
FPGA IMPLEMENTATIONS OF MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH LEVEL SYNTHESIS

Firas Abdul Ghani
Electronics, MS Thesis, 2017

Thesis Supervisor: Assoc. Prof. İlker HAMZAOĞLU

Keywords: HEVC, Fractional Interpolation, Motion Estimation, High-Level Synthesis

ABSTRACT

Joint collaborative team on video coding (JCT-VC) recently developed a new international video compression standard called High Efficiency Video Coding (HEVC). HEVC has 50% better compression efficiency than previous H.264 video compression standard. HEVC achieves this video compression efficiency by significantly increasing the computational complexity. Motion estimation is the most computationally complex part of video encoders. Integer motion estimation and fractional motion estimation account for 70% of the computational complexity of an HEVC video encoder. High-level synthesis (HLS) tools are started to be successfully used for FPGA implementations of digital signal processing algorithms. They significantly decrease design and verification time. Therefore, in this thesis, we proposed the first FPGA implementation of HEVC full search motion estimation using Vivado HLS. Then, we proposed the first FPGA implementations of two fast search (diamond search and TZ search) algorithms using Vivado HLS. Finally, we proposed the first FPGA implementations of HEVC fractional interpolation and motion estimation using Vivado HLS. We used several HLS optimization directives to increase performance and decrease area of these FPGA implementations.
HAREKET TAHMİNİ ALGORİTMALARININ VIVADO YÜKSEK SEVİYE SENTEZLEME İLE FPGA GERÇEKLEMELERİ

Firas Abdul Ghani
Elektronik Müh., Yüksek Lisans Tezi, 2017

Tez Danışmanı: Doç. Dr. İlker HAMZAOĞLU

Anahtar Kelimeler: HEVC, Kesirli Aradeğerleme, Hareket Tahmini, Yüksek Seviye Sentezleme

ÖZET

TABLE OF CONTENTS

ACKNOWLEDGEMENT ........................................................................................................ I

1 ABSTRACT ..................................................................................................................... II

2 ÖZET ............................................................................................................................. III

3 TABLE OF CONTENTS ............................................................................................... IV

LIST OF FIGURES .......................................................................................................... VI

LIST OF TABLES ................................................................................................................ VII

LIST OF ABBREVIATIONS .............................................................................................. VIII

1 CHAPTER I INTRODUCTION ....................................................................................... 1

1.1 HEVC Video Compression Standard ........................................................................ 1

1.2 High-Level Synthesis ............................................................................................... 3

1.3 Thesis Contributions ............................................................................................... 5

1.4 Thesis Organization .................................................................................................. 6

2 CHAPTER II FPGA IMPLEMENTATIONS OF INTEGER MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH-LEVEL SYNTHESIS .... 7

2.1 FPGA Implementation of HEVC Full Search Motion Estimation Algorithm Using Vivado High-LevelSynthesis ......................................................................................... 8

2.1.1 Full Search Motion Estimation Algorithm .......................................................... 8

2.1.2 FPGA implementation .........................................................................................

2.2 FPGA Implementation of Diamond Search Algorithm Using Vivado High-Level Synthesis ........................................................................................................... 11
2.2.1 Diamond Search Algorithm .................................................................
2.2.2 FPGA Implementation ......................................................................
2.3 FPGA Implementation of TZ Search Algorithm Using Vivado High-Level Synthesis .................................................................14
2.3.1 TZ Search Algorithm ........................................................................
2.3.2 FPGA Implementation ......................................................................
3 CHAPTER III FPGA IMPLEMENTATION OF FRACTIONAL MOTION ESTIMATION USING VIVADO HIGH-LEVEL SYNTHESIS ......................18
3.1 FPGA Implementations of HEVC Fractional Interpolation Using Vivado High-Level Synthesis ................................................................18
3.1.1 HEVC Fractional Interpolation Algorithm ...........................................
3.1.2 FPGA Implementations ......................................................................
3.2 FPGA Implementations of HEVC Fractional Motion Estimation Using High-Level Synthesis ................................................................................26
3.2.1 HEVC Fractional Motion Estimation Algorithm ....................................
3.2.2 FPGA Implementations ......................................................................
4 CHAPTER IV CONCLUSIONS AND FUTURE WORK .............................32
5 BIBLIOGRAPHY ......................................................................................33
LIST OF FIGURES

Figure 1.1 HEVC Encoder Block Diagram ................................................................. 2
Figure 1.2 HEVC Decoder Block Diagram ............................................................... 2
Figure 1.3 Xilinx Vivado HLS design flow .............................................................. 4
Figure 2.1 Integer Motion Estimation ......................................................................... 8
Figure 2.2 HEVC Full Search Motion Estimation HLS Implementation ..................... 10
Figure 2.3 Diamond Search Algorithm ..................................................................... 12
Figure 2.4 Diamond Search HLS Implementation ..................................................... 13
Figure 2.5TZS Diamond Search Pattern ..................................................................... 15
Figure 2.6 TZS Raster Search with Length 3 .............................................................. 15
Figure 2.7 TZ Search HLS Implementation ............................................................... 16
Figure 3.1 Integer, Half and Quarter Pixels ............................................................... 20
Figure 3.2 HEVC Fractional Interpolation HLS Implementation ............................... 21
Figure 3.3 Type A and Type B FIR Filters ............................................................... 24
Figure 3.4 Sub-pixel Search Locations ..................................................................... 26
Figure 3.5 HEVC fractional motion estimation HLS implementation ....................... 28
LIST OF TABLES

Table 1.1 Xilinx Vivado HLS Optimizations..................................................................................4
Table 2.1 Full Search Motion Estimation HLS implementation Results ................................10
Table 2.2 HEVC Full Search Motion Estimation HLS Implementation Results ..................11
Table 2.3 HEVC Full Search Motion Estimation With Variable Search Range ....................11
Table 2.4 Diamond Search HLS Implementation Results .........................................................14
Table 3.1 HLS Implementation without Manual Loop Unrolling with Multipliers Results ..23
Table 3.2 HLS Implementation with Multipliers Results ..........................................................23
Table 3.3 HLS Implementation with Adders and Shifters Results ........................................23
Table 3.4 HLS Implementation with MCM Results .................................................................24
Table 3.5 HEVC Fractional Interpolation Hardware Comparison ........................................25
Table 3.6 HEVC Fractional Motion Estimation HLS Implementation Results ...................30
Table 3.7 Allocation Analysis for MM HLS Implementations ...............................................30
Table 3.8 HEVC Fractional Motion Estimation Hardware Comparison ...............................31
**LIST OF ABBREVIATIONS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>Block RAM</td>
</tr>
<tr>
<td>CABAC</td>
<td>Context Adaptive Binary Arithmetic Coding</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DST</td>
<td>Discrete Sine Transform</td>
</tr>
<tr>
<td>DVI</td>
<td>Digital Visual Interface</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>HEVC</td>
<td>High Efficiency Video Coding</td>
</tr>
<tr>
<td>HM</td>
<td>HEVC Test Model</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal to Noise Ratio</td>
</tr>
<tr>
<td>PU</td>
<td>Prediction Unit</td>
</tr>
<tr>
<td>SAO</td>
<td>Sample Adaptive Offset</td>
</tr>
<tr>
<td>TU</td>
<td>Transform Unit</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>HLS</td>
<td>High Level Synthesis</td>
</tr>
<tr>
<td>SAD</td>
<td>Sum Of Absolute Difference</td>
</tr>
<tr>
<td>TZA</td>
<td>TZ Search Algorithm</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

1.1 HEVC Video Compression Standard

Since better coding efficiency is required for high resolution videos, Joint Collaborative Team on Video Coding (JCT-VC) recently developed a new video compression standard called High Efficiency Video Coding (HEVC) [1, 2, 3]. HEVC provides 50% better coding efficiency than previous H.264 video compression standard. HEVC also provides 23% bit rate reduction for the intra prediction only case [4]. The video compression efficiency achieved in HEVC standard is not a result of any single feature but rather a combination of a number of encoding tools such as intra prediction, motion estimation, deblocking filter and entropy coder. Motion estimation is the most computationally complex part of video encoders. Integer motion estimation and fractional motion estimation account for 70% of the computational complexity of an HEVC video encoder.

The top-level block diagram of an HEVC encoder and decoder are shown in Figure 1.1 and Figure 1.2, respectively. An HEVC encoder has a forward path and a reconstruction path. The forward path is used to encode a video frame by using intra and inter predictions and to create the bit stream after the transform and quantization process. Reconstruction path in the encoder ensures that both encoder and decoder use identical reference frames for intra and inter prediction because a decoder never gets original images.
In the forward path, frame is divided into coding units (CU) that can be an 8x8, 16x16, 32x32 or 64x64 pixel block. Each CU is encoded in intra or inter mode depending on the mode decision. Intra and inter prediction processes use prediction unit (PU) partitioning inside the CUs. Prediction unit (PU) sizes can be from 4x4 up to 64x64. Mode decision determines whether a PU will be coded intra or inter mode based on video quality and bit-rate. After mode decision determines the prediction mode, predicted block is subtracted from original block, and residual data is generated. Then, residual data transformed by discrete cosine transform (DCT) and quantized. Transform unit (TU) sizes can be from 4x4 up to 32x32. Finally, entropy coder generates the encoded bitstream.
Reconstruction path begins with inverse quantization and inverse transform operations. The quantized transform coefficients are inverse quantized and inverse transformed to generate the reconstructed residual data. Since quantization is a lossy process, inverse quantized and inverse transformed coefficients are not identical to the original residual data. The reconstructed residual data are added to the predicted pixels in order to create the reconstructed frame. DBF is, then, applied to reduce the effects of blocking artifacts in the reconstructed frame.

### 1.2 High-Level Synthesis

Recently, high-level synthesis (HLS) tools started to generate production quality register transfer level (RTL) implementations from high-level specifications. HLS tools improve productivity of hardware designers by reducing both design and verification time.

In this thesis, Xilinx Vivado HLS tool is used. It is one of the successful commercial HLS tools. It takes C, C++ or SystemC codes as input, and generates Verilog or VHDL codes. Design flow used in this thesis for the FPGA implementations of motion estimation algorithms using Xilinx Vivado HLS is shown in Figure 1.3. First, software models of HEVC video compression algorithms are developed using HEVC reference software video encoder (HM) 15.0 [5]. After the software models are verified with HEVC test sequences, C codes for HLS are developed. Then, the C codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool. Several optimizations offered by Xilinx Vivado HLS tool are also used to increase performance and decrease area of the proposed FPGA implementations. The Verilog RTL codes are synthesized and mapped to a Xilinx Virtex 6 FPGA using Xilinx ISE 14.7. Finally, the FPGA implementations are verified with post place and route simulations.

Xilinx Vivado HLS tool provides C specification testbench to verify the code. This C testbench is used by the tool to verify that the functionality of the synthesized RTL is same as the functionality of the original C code. After verifying the functionality with C testbench, Vivado HLS tool generates hardware (Verilog or VHDL) testbench to verify the hardware. Then, HLS tool compares the output of C testbench and hardware testbench. If they are same, it indicates that the hardware is verified.
Xilinx Vivado HLS tool performs scheduling of operations, allocation of registers, and binding of operations to functional units. Xilinx Vivado HLS tool provides many optimizations (pragmas) for scheduling, allocation and binding. It also provides bit-accurate or cycle-accurate implementations. It allows adding specific RAM blocks, FIFOs, ROMs or specific DSP blocks. In addition it generates I/O interfaces to connect hardware modules with memories or other peripherals. Xilinx Vivado HLS tool offers these optimizations to increase performance and decrease area of HLS implementations. These optimizations can be grouped as shown in Table 1.1.
Loop Unrolling (LU) directive is used to increase performance using more resources. It creates multiple copies of loop body, and compute them in parallel. In this way, it decreases the loop iterations and increases the performance. However, loop unrolling may cause memory access problems in HLS designs.

Allocation (ALC) directive is used to specify the maximum number of resources that can be used in hardware. It forces the HLS tool to perform resource sharing. It therefore decreases the hardware area. Allocation can be used for addition, subtraction, multiplication, division, shift and comparison operations.

Pipeline (PIPE) directive performs pipelining to increase the performance. Xilinx Vivado HLS tool performs pipelining automatically. However, number of pipeline stages can also be defined for further performance increase.

Resource (RES) directive is used to specify which resource will be used to implement a variable such as an array, arithmetic operation or function argument. DSP elements, specific RAM blocks, FIFOs or ROMs can be used with resource directive.

Array map (AMAP) directive is used to map multiple small arrays into a single large array. The large array can be targeted to a single large memory (RAM or FIFO) resource. It is also used to control how (horizontal or vertical) data is stored in BRAMs.

Array partition (APAR) directive partitions the large arrays into multiple smaller arrays or individual registers for parallel data accesses.

Xilinx Vivado HLS tool also provides a specific library for designing bit-accurate (BIT) models in C codes.

A few HLS implementations for HEVC video compression standard are proposed in the literature [6]-[8]. A few HLS implementations for H.264 video compression standard are proposed in the literature [9]-[12]. There are a few HLS implementations based on MPEG reconfigurable video coding [13]-[14]. There are several HLS implementations for image and video processing algorithms such as sorting in the median filter [15]-[18].

1.3 Thesis Contributions

In this thesis, we proposed the first FPGA implementation of HEVC full search motion algorithm using HLS in the literature. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC reference software video encoder (HM) version 15 [5]. We used several optimizations offered by Vivado HLS to achieve real-time performance. The proposed FPGA implementation of HEVC full search
motion estimation algorithm using HLS can process 30 full HD video frames per second for all PU sizes and for fixed search range (64x64). It can process 29 full HD frames per second for variable search ranges.

Fast search motion estimation algorithms are used to reduce computational complexity of motion estimation. Diamond Search (DS) and TZ Search (TZS) are very successful fast search motion estimation algorithms. Therefore, in this thesis, first FPGA implementations of DS and TZS algorithms using HLS in the literature are proposed. The proposed DS and TZS motion estimation FPGA implementations can process 127 full HD (1920x1080) and 46 full HD video frames per second, respectively.

We also proposed the first FPGA implementation of HEVC fractional interpolation and motion estimation using HLS in the literature. We used several optimizations offered by Vivado HLS to achieve real-time performance. The proposed HEVC fractional interpolation and HEVC fractional motion estimation FPGA implementations can process 45 quad full HD (3840x2160) and 46 full HD video frames per second, respectively.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter II first explains FPGA implementations of HEVC full search motion estimation algorithm using Vivado HLS and presents the experimental results. It, then, explains FPGA implementations of two fast search (Diamond Search and TZ Search) motion estimation algorithms using Vivado HLS and presents the experimental results.

Chapter III explains FPGA implementations of HEVC fractional interpolation and fractional motion estimation algorithms using Vivado HLS and presents the experimental results.

Chapter IV presents conclusions and future work.
CHAPTER II

FPGA IMPLEMENTATIONS OF INTEGER MOTION ESTIMATION ALGORITHMS USING VIVADO HIGH-LEVEL SYNTHESIS

Motion estimation (ME) is used to remove temporal redundancy between current frame and reference frame that has been encoded previously. As shown in Figure 2.1, integer motion estimation (IME) divides the current frame into blocks and finds the motion vector (MV) for each block by determining the reference block in the reference frame that gives the smallest sum of absolute difference (SAD) for this block. Then, it calculates the difference between the current block and the best matching reference block, and encodes this residual and the motion vector.

HEVC standard divides the current frame into blocks called Prediction Units (PUs) for IME. In HEVC standard, 24 different PU sizes are defined. These PU sizes range from 4x8 or 8x4 to 64x64. This allows HEVC standard to do better compression than previous video compression standards.
Figure 2.1 Integer Motion Estimation

2.1 FPGA Implementation of HEVC Full Search Motion Estimation Algorithm Using Vivado High-Level Synthesis

2.1.1 Full Search Motion Estimation Algorithm

Full Search (FS) algorithm exhaustively searches all search locations in the defined search window in the reference frame. Therefore, it finds the best MV in the search window. However, it is the most computationally complex motion estimation algorithm.

FS algorithm calculates the SAD value for each search location as shown in Equation 2.1.

\[
SAD = \sum_{i=0}^{m} \sum_{j=0}^{n} |R_{ij} - C_{ij}|
\]  

(2.1)

R is a pixel in the reference frame. C is a pixel in the current frame. It determines the search location with the minimum SAD value and the MV corresponding to this search location.
2.1.2 FPGA implementation

We, first, designed a full search IME hardware for fixed current block size (8x8) and fixed search range (16x16). In this hardware, 8 parallel absolute difference hardware calculate absolute differences for one column of 8x8 PU. After 8 iterations, SAD value is calculated by adding absolute difference values. 16x16 array stores all SAD values for comparison. Then, comparison unit compares SAD values, and determines the minimum SAD value and the corresponding motion vector.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.1.

PIPE, LU, APAR and RES directives are used to increase the performance. Number of frames per second processed by this FPGA implementation is calculated as shown in Equation (2.2).

\[
\text{Frequency (MHz)} \times 1000000 \times \left( \frac{\text{Frame Size}}{\text{Search Range Size}} \right) \times \text{Clock Cycles}\]

(2.2)

Then, a full search IME hardware implementing the FS IME algorithm in HEVC reference software video encoder (HM) version 15 [5] is designed. It supports all 24 PU sizes defined in HEVC standard. It implements 64x64 fixed search range. The proposed hardware is shown in Fig. 2.2.

In HEVC, 593 SADs and 593 MVs should be calculated for all PU sizes. Numbers of SADs and MVs that should be calculated for each PU size are as follows: 4x8 (128 SADs and 128 MVs), 8x4 (128 SADs and 128 MVs), 8x8 (64 SADs and 64 MVs), 4x16 (64 SADs and 64 MVs), 8x16 (32 SADs and 32 MVs), 12x16 (20 SADs and 20 MVs), 16x4 (64 SADs and 64 MVs), 16x12 (20 SADs and 20 MVs), 16x16 (16 SADs and 16 MVs), 8x32 (16 SADs and 16 MVs), 16x32 (8 SADs and 8 MVs), 24x32 (4 SADs and 4 MVs), 32x8 (16 SADs and 16 MVs), 32x16 (8 SADs and 8 MVs), 32x24 (4 SADs and 4 MVs), 32x32 (4 SADs and 4 MVs), 16x64 (4 SADs and 4 MVs), 32x64 (2 SADs and 2 MVs), 48x64 (1 SAD and 1 MV), 64x16 (4 SADs and 4 MVs), 64x32 (2 SADs and 2 MVs), 64x48 (1 SAD and 1 MV) and 64x64 (1 SAD and 1 MV).
Table 2.1 Full Search Motion Estimation HLS implementation Results
For 8x8 PU Size

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (8x8 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>290</td>
<td>644</td>
<td>490</td>
<td>2</td>
<td>267</td>
<td>2031</td>
<td>0.912</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2560*144</td>
<td></td>
</tr>
<tr>
<td>APAR_RES(BRAM)_PIPE_LU</td>
<td>2132</td>
<td>6247</td>
<td>2346</td>
<td>49</td>
<td>200</td>
<td>256</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2560*144</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.2** HEVC Full Search Motion Estimation HLS Implementation

First, reference and current pixels are stored into integer pixels buffer. 128x128 reference pixels are stored in order to be able to search all search locations in the 64x64 search range. Then, SAD values for 4x4 PUs in the 64x64 CU are calculated. Since there are 16x16 4x4 PUs in the 64x64 CU, a 16x16 array is used to store SAD values of 4x4 PUs. Then, SAD values for the other PU sizes are calculated by adding the SAD values of 4x4 PUs. After that, comparison unit compares the SAD values, determines the 593 minimum SAD values for all PU sizes and their corresponding MVs, and stores them into two different arrays.

APAR is used for the 16x16 array storing SADs for 4x4 PUs. In this way, latency of calculating SAD values of larger PUs is reduced by accessing the SAD values of 4x4 PUs in parallel. Loop unrolling (LU) is used to perform absolute difference calculations...
in parallel. PIPE is used to increase the performance. Bit-accurate model is used in order to decrease adder bit-width.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.2.

Finally, this HLS implementation is parametrized to support 4 different (8x8, 16x16, 32x32 and 64x64) search ranges by only changing the boundaries of nested loops calculating SAD values according to the selected search range.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.3.

### Table 2.2 HEVC Full Search Motion Estimation HLS Implementation Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (64x64 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>6858</td>
<td>17632</td>
<td>18397</td>
<td>6</td>
<td>0</td>
<td>125</td>
<td>105676</td>
<td>0.23</td>
</tr>
<tr>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>29875</td>
<td>88286</td>
<td>76271</td>
<td>138</td>
<td>0</td>
<td>86</td>
<td>5705</td>
<td>30</td>
</tr>
</tbody>
</table>

### Table 2.3 HEVC Full Search Motion Estimation With Variable Search Range HLS Implementation Results

<table>
<thead>
<tr>
<th>Search Range</th>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq.</th>
<th>Clock Cycles</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>34302</td>
<td>87259</td>
<td>76345</td>
<td>138</td>
<td>0</td>
<td>83</td>
<td>441</td>
<td>372 FHD</td>
</tr>
<tr>
<td>16x16</td>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>809</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>809</td>
<td>202 FHD</td>
</tr>
<tr>
<td>32x32</td>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>1929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1929</td>
<td>85 FHD</td>
</tr>
<tr>
<td>64x64</td>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>5705</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5705</td>
<td>29 FHD</td>
</tr>
</tbody>
</table>

### 2.2 FPGA Implementation of Diamond Search Algorithm Using Vivado High-Level Synthesis

Fast search motion estimation algorithms are used to reduce computational complexity of FS algorithm at the expense of slight PSNR loss and bitrate increase.
2.2.1 Diamond Search Algorithm

Diamond search (DS) motion estimation algorithm follows a diamond search pattern. DS algorithm has two steps; large diamond search (LDS) and small diamond search (SDS). LDS calculates SAD values for 9 search locations that form a large diamond shape as shown in Figure 2.3 (a), and determines the search location with minimum SAD. If the search location with minimum SAD is at the center of the diamond shape, SDS is performed. Otherwise, LDS is performed around the search location with minimum SAD as shown in Figure 2.3 (c). SDS calculates SAD values for 4 search locations that form a small diamond shape as shown in Figure 2.3 (b), and determines the search location with minimum SAD and the corresponding motion vector.

![Diamond Search Algorithm](image)

**Figure 2.3** Diamond Search Algorithm

2.2.2 FPGA Implementation

The proposed DS HLS implementation for fixed current block size (64x64) and fixed search range size (64x64) is shown in Figure 2.4. First, pixels in the current block in the current frame and necessary pixels in the reference frame are stored into integer pixels buffers. In order to decrease memory area, only 68x68 reference pixels are stored. After the first LDS, if another LDS is performed, only new reference pixels are read and stored into integer pixels buffer. Other reference pixels are shifted.
LDS may never find a search location with minimum SAD that is at the center of the diamond shape. Therefore, a maximum number of LDS allowed should be defined. In the proposed HLS implementation, this maximum number is defined as a parameter which can be between 1 and 10.

In the proposed HLS implementation, 9 SAD values that should be calculated for LDS are calculated in parallel. 64 parallel absolute difference hardware are used for calculating each SAD value. Then, comparison unit determines the search location with minimum SAD.

If the search location with minimum SAD is at the center of the diamond shape, SDS is performed. Otherwise, LDS is performed around the search location with minimum SAD. However, if the maximum number of LDSs allowed are performed, SDS is performed instead of LDS. If another LDS is performed, only new reference pixels are read and stored into integer pixels buffer. Other reference pixels are shifted. We used loop unrolling for shifting.

In the proposed HLS implementation, 4 SAD values that should be calculated for SDS are calculated in parallel. Then, comparison unit determines the search location with minimum SAD. Finally, the minimum SAD values found in LDS and SDS are
compared, and the minimum SAD value and the corresponding MV for DS are determined.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.4.

APAR, RES, PIPE, LU optimization directives are used in order to increase the performance and decrease the hardware area. Bit-accurate model is also used to decrease hardware area. Number of clock cycles changes with number of steps. These results show that the proposed DS HLS implementation can process 127 full HD frames per second.

Table 2.4 Diamond Search HLS Implementation Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (64x64 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>10132</td>
<td>28322</td>
<td>16535</td>
<td>4</td>
<td>0</td>
<td>108</td>
<td>4754</td>
<td>46352</td>
</tr>
<tr>
<td>APAR_RES(BRAM)_PIPE_LU_BIT</td>
<td>12573</td>
<td>37457</td>
<td>20859</td>
<td>67</td>
<td>0</td>
<td>139</td>
<td>334</td>
<td>2152</td>
</tr>
</tbody>
</table>

2.3 FPGA Implementation of TZ Search Algorithm Using Vivado High-Level Synthesis

2.3.1 TZ Search Algorithm

TZ search (TZS) is another fast search motion estimation algorithm. It finds better MVs than DS. But, it has higher computational complexity than DS. TZS uses two different search patterns; diamond search pattern and raster search pattern as shown in Figure 2.5 and Figure 2.6, respectively. Raster search is similar to full search, but it searches less number of search locations. It is used as a refinement after the diamond search pattern.

Diamond search pattern starts searching at the (0,0) search location, and it proceeds according to the steps shown in Figure 2.5. It calculates the SAD values and determines the minimum SAD in each step. It has two termination conditions. The first one is reaching the search window boundaries. The second one is not finding minimum SAD in three consecutive steps. For example, if the SAD value of (0,0) search location
is smaller than all SAD values calculated in steps 1, 2, and 4, then it is terminated, and the SAD value of (0,0) search location is determined as the minimum SAD.

![Figure 2.5 TZS Diamond Search Pattern](image)

**Figure 2.5 TZS Diamond Search Pattern**

![Figure 2.6 TZS Raster Search with Length 3](image)

**Figure 2.6 TZS Raster Search with Length 3**

### 2.3.2 FPGA implementation

The proposed TZS HLS implementation for fixed current block size (64x64) and fixed search range size (64x64) is shown in Figure 2.7. First, 64x64 current pixels and 128x128 reference pixels are stored into integer pixels buffers. Then, diamond search pattern is performed. Since the search range size is 64x64, maximum number of steps for the diamond search pattern is 6. In each step, SAD values for the search locations are calculated and the search location with minimum SAD is determined.
As shown in Figure 2.5, number of search locations for all the steps after step 1 is 8. In order not to repeat the same operations for SAD calculations in steps 2, 3, 4, 5, and 6, control variables are added to HLS code to update memory addresses after each step.

After each step, control unit checks the termination conditions of diamond search pattern. If a termination condition occurs, diamond search pattern is terminated. In that case, if starting condition of raster search pattern occurs, raster search pattern is performed.

As shown in Figure 2.6, raster search pattern is similar to full search. However, it skips some search locations based on raster search length. It searches only the search locations shown as black in the figure. SAD values of these search locations are calculated and the search location with minimum SAD is determined.

Finally, comparison unit compares the minimum SAD found in diamond search pattern and the minimum SAD found in raster search pattern, and determines the minimum SAD and the corresponding MV.

Verilog RTL codes generated by Xilinx Vivado HLS tool for this HLS implementation are verified with post place and route simulations. The implementation results are shown in Table 2.5.
APAR, RES, PIPE, LU optimization directives are used in order to increase the performance and decrease the hardware area. Bit-accurate model is also used to decrease hardware area. These results show that the proposed TZS HLS implementation can process 46 full HD frames per second.

Table 2.5 TZ Search HLS Implementation Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (64x64 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>9744</td>
<td>25723</td>
<td>15821</td>
<td>10</td>
<td>0</td>
<td>128</td>
<td>66321</td>
<td>4 1920x1080</td>
</tr>
<tr>
<td>APAR_RES(BRAM), PIPE_LU_BIT</td>
<td>39406</td>
<td>114412</td>
<td>15943</td>
<td>128</td>
<td>0</td>
<td>92</td>
<td>3980</td>
<td>46 1920x1080</td>
</tr>
</tbody>
</table>
CHAPTER III
FPGA IMPLEMENTATION OF FRACTIONAL MOTION
ESTIMATION USING VIVADO HIGH-LEVEL SYNTHESIS

In order to increase the performance of integer pixel motion estimation, fractional motion estimation (FME), which provides half and quarter pixel accurate motion vector (MV) refinement, is performed. First, fractional interpolation is performed to generate fractional pixels. Then, fractional motion estimation is performed using fractional pixels.

Fractional (half-pixel and quarter-pixel) interpolation is one of the most computationally intensive parts of HEVC video encoder and decoder. On average, one fourth of the HEVC encoder complexity and 50% of the HEVC decoder complexity are caused by fractional interpolation [19]. FME is heavily used in an HEVC encoder. It accounts for up to 49% of total encoding time of HEVC video encoder [20].

HEVC uses FME same as H.264. However, HEVC FME has higher computational complexity than H.264 FME. HEVC standard uses three different 8-tap FIR filters for fractional interpolation and up to 64×64 prediction unit (PU) sizes [21].

3.1 FPGA Implementations of HEVC Fractional Interpolation Using Vivado High-Level Synthesis

Since HEVC fractional interpolation algorithm uses FIR filters, it is suitable for HLS implementation. Therefore, in this thesis, the first FPGA implementation of HEVC fractional interpolation algorithm using Xilinx Vivado HLS tool in the literature is
The proposed HEVC fractional interpolation hardware is implemented on Xilinx FPGAs using Xilinx Vivado HLS tool. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5].

Three HEVC fractional interpolation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

Some of the optimization options of Xilinx Vivado HLS tool are used in order to increase performances of the FPGA implementations such as pipelining, allocation, resource optimizations, array mapping and array partitioning. Verilog RTL codes generated by Xilinx Vivado HLS tool for the three HEVC fractional interpolation HLS implementations are verified to work in a Xilinx Virtex 6 FPGA.

Using HLS tool significantly reduced the FPGA development time. The implementation results show that the proposed HEVC fractional interpolation FPGA implementation, in the worst case, can process 45 quad full HD (3840x2160) video frames per second with acceptable hardware area.

The HEVC fractional interpolation HLS implementation proposed in this thesis is the first HLS implementation for HEVC fractional interpolation algorithm in the literature. In Section 3.1.2, it is compared with the handwritten HEVC fractional interpolation hardware implementations proposed in the literature [23]-[27].

### 3.1.1 HEVC Fractional Interpolation Algorithm

In HEVC standard, 3 different 8-tap FIR filters are used for both half-pixel and quarter-pixel interpolations. These 3 FIR filters type A, type B and type C are shown in (3.1), (3.2), and (3.3), respectively. The shift1 value is determined based on bit depth of the pixel.
Figure 3.1 Integer, Half and Quarter Pixels

![Image of pixel types]

<table>
<thead>
<tr>
<th>Integer Pixel</th>
<th>Half Pixel</th>
<th>Quarter Pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{x,y}$</td>
<td>$a_{x,y}$</td>
<td>$e_{x,y}$</td>
</tr>
<tr>
<td>$b_{x,y}$</td>
<td></td>
<td>$f_{x,y}$</td>
</tr>
<tr>
<td>$c_{x,y}$</td>
<td></td>
<td>$g_{x,y}$</td>
</tr>
<tr>
<td>$d_{x,y}$</td>
<td>$h_{x,y}$</td>
<td>$i_{x,y}$</td>
</tr>
<tr>
<td>$n_{x,y}$</td>
<td></td>
<td>$j_{x,y}$</td>
</tr>
<tr>
<td>$p_{x,y}$</td>
<td></td>
<td>$k_{x,y}$</td>
</tr>
<tr>
<td>$q_{x,y}$</td>
<td></td>
<td>$l_{x,y}$</td>
</tr>
</tbody>
</table>

Integer pixels ($A_{x,y}$), half pixels ($a_{x,y}$, $b_{x,y}$, $c_{x,y}$, $d_{x,y}$, $h_{x,y}$, $n_{x,y}$) and quarter pixels ($e_{x,y}$, $f_{x,y}$, $g_{x,y}$, $i_{x,y}$, $j_{x,y}$, $k_{x,y}$, $p_{x,y}$, $q_{x,y}$, $r_{x,y}$) in a PU are shown in Figure 3.1. The half pixels $a$, $b$, $c$ are interpolated from nearest integer pixels in horizontal direction using type A, type B and type C filters, respectively. The half-pixels $d$, $h$, $n$ are interpolated from nearest integer pixels in vertical direction using type A, type B and type C filters, respectively. The quarter pixels $e$, $f$, $g$ are interpolated from the nearest $a$, $b$, $c$ half pixels respectively in vertical direction using type A filter. The quarter pixels $i$, $j$, $k$ are interpolated similarly using type B filter. The quarter pixels $p$, $q$, $r$ are interpolated similarly using type C filter.

HEVC fractional interpolation algorithm used in HEVC encoder calculates all the fractional pixels necessary for the fractional motion estimation.

### 3.1.2 FPGA Implementations

The proposed HLS implementation of HEVC fractional interpolation is shown in Figure 3.2. The proposed HLS implementation is synthesized to Verilog RTL using...
Xilinx Vivado HLS tool. The C codes given as input to Xilinx Vivado HLS tool are developed based on the HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5].

In the proposed HLS implementation, half pixels and quarter pixels for an 8x8 PU are calculated using 15x15 integer pixels. Half pixels and quarter pixels for larger PU sizes can be calculated by calculating the half pixels and quarter pixels for each 8x8 part of a PU separately. In the C codes, 15 integer pixels are taken as input in each clock cycle. 8 a, 8 b and 8 c half-pixels are interpolated in parallel in each clock cycle. 15x8 a, 15x8 b, and 15x8 c half pixels are interpolated in 15 clock cycles, and they are stored into registers for quarter pixel interpolation. In the same 15 clock cycles, 15x8 integer pixels are also stored into registers for interpolating d, h, n half pixels. Then, 8x8 d, 8x8 h, 8x8 n half pixels are interpolated using 15x8 integer pixels. Finally, all quarter pixels (e, f, g, i, j, k, p, q, r) are interpolated using 15x8 a, 15x8 b, and 15x8 c half pixels.

Three HEVC fractional interpolation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

Figure 3.2 HEVC Fractional Interpolation HLS Implementation
Verilog RTL codes generated by Xilinx Vivado HLS tool for these three HLS implementations are verified with RTL simulations. RTL simulation results matched the results of HEVC fractional interpolation software implementation in the HEVC reference software video encoder (HM) version 15 [5]. The Verilog RTL codes are synthesized and mapped to a Xilinx XC6VLX550T FF1760 FPGA with speed grade 2 using Xilinx ISE 14.7. The FPGA implementations are verified with post place and route simulations.

We used several optimizations offered by Xilinx Vivado HLS tool to increase the performance and decrease the area of the proposed HLS implementations [28]. We tried to use loop unrolling directive. However, loop unrolling directive did not work correctly for the proposed HLS implementations. In [29], it is mentioned that loop unrolling may cause memory access problems in HLS designs, and current generation of HLS tools may ignore these problems. As shown in Table 3.1, the performance of the HLS implementation, which implements multiplications with constants using multiplication operations, without loop unrolling is very low. Therefore, we performed manual loop unrolling in the proposed HLS implementations to increase their performances.

In the proposed HLS implementations, ALC is used for subtraction, addition, multiplication, and shifting operations. PIPE directive is used in the proposed HLS implementations. In the proposed HLS implementations, RES directive is used to store input integer pixels into BRAMS. In the proposed HLS implementations, AMAP directive is used to control how data is stored in BRAMS so that the number of BRAMS used in the hardware is reduced as much as possible. In the proposed HLS implementations, APAR directive is used to partition the arrays that store a, b, and c half pixels to increase quarter pixel interpolation performance. In the proposed HLS implementations, bit accurate (BIT) model is used to decrease adder bit widths and therefore hardware area.

The FPGA implementation results for the first HLS implementation (MM) are given in Table 3.2. In this HLS implementation, in the C codes, multiplications with constants are implemented using multiplication operations. These multiplication operations are mapped to DSP48 blocks in RTL synthesis. This decreased the number of LUTs and DFFs used in the hardware. Allocation (ALC), pipeline (PIPE), resource (RES) and array map (AMAP) directives are used in this HLS implementation. In the table, M shows the number of multipliers used in the ALC directive.
The FPGA implementation results for the second HLS implementation (MAS) are given in Table 3.3. In this HLS implementation, multiplications with constants are implemented using addition and shift operations. This HLS implementation does not use any DSP48 blocks, but it uses more LUTs and DFFs than MM. It also has higher performance than MM. Pipeline (PIPE), resource (RES) and array map (AMAP) directives are used in this HLS implementation.

Table 3.1 HLS Implementation without Manual Loop Unrolling with Multipliers Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (8x8 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>885</td>
<td>2565</td>
<td>1411</td>
<td>1</td>
<td>15</td>
<td>250</td>
<td>1921</td>
<td>1/3840x2160</td>
</tr>
</tbody>
</table>

Table 3.2 HLS Implementation with Multipliers Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (8x8 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>4623</td>
<td>14110</td>
<td>7526</td>
<td>0</td>
<td>113</td>
<td>200</td>
<td>156</td>
<td>10/3840x2160</td>
</tr>
<tr>
<td>ALC(M128)</td>
<td>4769</td>
<td>14133</td>
<td>6226</td>
<td>0</td>
<td>135</td>
<td>168</td>
<td>148</td>
<td>9/3840x2160</td>
</tr>
<tr>
<td>PIPE</td>
<td>4938</td>
<td>14086</td>
<td>8736</td>
<td>0</td>
<td>113</td>
<td>201</td>
<td>56</td>
<td>28/3840x2160</td>
</tr>
<tr>
<td>RES(BRAM)</td>
<td>4723</td>
<td>13883</td>
<td>7395</td>
<td>4</td>
<td>113</td>
<td>201</td>
<td>156</td>
<td>10/3840x2160</td>
</tr>
<tr>
<td>ALC(M128)_RES(BRAM)_PIPE</td>
<td>5197</td>
<td>14366</td>
<td>8000</td>
<td>4</td>
<td>147</td>
<td>167</td>
<td>56</td>
<td>23/3840x2160</td>
</tr>
<tr>
<td>ALC(M128)_AMAP(4)_RES(BRAM)_PIPE</td>
<td>4299</td>
<td>12401</td>
<td>7964</td>
<td>2</td>
<td>147</td>
<td>167</td>
<td>56</td>
<td>23/3840x2160</td>
</tr>
<tr>
<td>ALC(M20)_AMAP(4)_RES(BRAM)_PIPE</td>
<td>4299</td>
<td>13100</td>
<td>8037</td>
<td>2</td>
<td>59</td>
<td>168</td>
<td>56</td>
<td>23/3840x2160</td>
</tr>
</tbody>
</table>

Table 3.3 HLS Implementation with Adders and Shifters Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (8x8 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>4809</td>
<td>15629</td>
<td>9095</td>
<td>0</td>
<td>0</td>
<td>202</td>
<td>133</td>
<td>12/3840x2160</td>
</tr>
<tr>
<td>AMAP(4)_RES(BRAM)_PIPE</td>
<td>4891</td>
<td>15716</td>
<td>9436</td>
<td>2</td>
<td>0</td>
<td>200</td>
<td>55</td>
<td>28/3840x2160</td>
</tr>
</tbody>
</table>

The FPGA implementation results for the last HLS implementation (MMCM) are given in Table 3.4. In this HLS implementation, addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22]. The type A and type B FIR filter equations for 8 a half pixels and 8 b half pixels are shown in Figure 3.3. As shown in Figure 3.3, common sub-expressions are calculated in...
different equations and same integer pixel is multiplied with different constant coefficients in different equations. Therefore, in this HLS implementation, common sub-expressions in different FIR filter equations are calculated once, and the result is used in all the equations. This HLS implementation also uses Hcub MCM algorithm in order to reduce number and size of the adders, and to minimize the adder tree depth [22]. Hcub algorithm tries to minimize number of adders, their bit size and adder tree depth in a multiplier block, which multiplies a single input with multiple constants. This HLS implementation has the best performance with acceptable hardware area. Allocation (ALC), pipeline (PIPE), array partition (APAR) directives and bit-accurate (BIT) model are used in this HLS implementation. In the table, A and S show the number of adders and subtractors used in the ALC directive, respectively.

Table 3.4 HLS Implementation with MCM Results

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq. (MHz)</th>
<th>Clock Cycles (8x8 PU)</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPT</td>
<td>4850</td>
<td>15632</td>
<td>6673</td>
<td>2</td>
<td>0</td>
<td>201</td>
<td>195</td>
<td>8</td>
</tr>
<tr>
<td>ALC(A500_S500)_PIPE</td>
<td>5288</td>
<td>14619</td>
<td>10118</td>
<td>0</td>
<td>0</td>
<td>168</td>
<td>29</td>
<td>45</td>
</tr>
<tr>
<td>ALC(A500_S500)_PIPE_BIT</td>
<td>4426</td>
<td>14225</td>
<td>9984</td>
<td>0</td>
<td>0</td>
<td>168</td>
<td>29</td>
<td>45</td>
</tr>
</tbody>
</table>

![Figure 3.3 Type A and Type B FIR Filters](image)

The best HEVC fractional interpolation HLS implementation proposed in this thesis (MMCM with ALC(A500_S500)_APAR_PIPE_BIT) is compared with the handwritten HEVC fractional interpolation hardware implementations proposed in the literature [23]-[27]. The comparison results are shown in Table 3.5.

The proposed MMCM HLS implementation is similar to the handwritten HEVC fractional interpolation hardware implementation proposed in [23]. In [23], common
sub-expressions in different FIR filter equations are calculated once, and the result is used in all the equations. Also, addition and shift operations are implemented using Hcub multiplierless constant multiplication (MCM) algorithm.

In [23], the handwritten Verilog RTL codes are synthesized and mapped to a Xilinx XC6VLX130T FF1156 FPGA with speed grade 3. In this thesis, the handwritten Verilog RTL codes proposed in [23] are synthesized and mapped to a Xilinx XC6VLX550T FF1760 FPGA with speed grade 2 for fair comparison with the proposed MMCM HLS implementation. The proposed MMCM HLS implementation has higher performance than the handwritten HEVC fractional interpolation hardware implementation proposed in [23] at the expense of larger area.

<table>
<thead>
<tr>
<th>Technology</th>
<th>[23]</th>
<th>[24]</th>
<th>[25]</th>
<th>[26]</th>
<th>[27]</th>
<th>Proposed (MMCM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>Xilinx Virtex 6</td>
<td>90 nm</td>
<td>90 nm</td>
<td>150 nm</td>
<td>90 nm</td>
<td>130 nm</td>
</tr>
<tr>
<td>Gate/Slice Count</td>
<td>1597</td>
<td>28.5 K</td>
<td>32.5 K</td>
<td>30.2 K</td>
<td>224 K</td>
<td>126.8 K</td>
</tr>
<tr>
<td>Max Speed (MHz)</td>
<td>200</td>
<td>200</td>
<td>171</td>
<td>312</td>
<td>333</td>
<td>208</td>
</tr>
<tr>
<td>Frames per Second</td>
<td>3840x2160</td>
<td>3840x2160</td>
<td>3840x2160</td>
<td>3840x2160</td>
<td>1920x1080</td>
<td>3840x2160</td>
</tr>
<tr>
<td>Design</td>
<td>ME + MC</td>
<td>ME + MC</td>
<td>Only MC</td>
<td>ME + MC</td>
<td>ME + MC</td>
<td>ME + MC</td>
</tr>
</tbody>
</table>

Since the handwritten HEVC fractional interpolation hardware implementation proposed in [24] is designed only for motion compensation (MC), it has higher performance and lower area than the proposed MMCM HLS implementation.

The handwritten HEVC fractional interpolation hardware implementation proposed in [25] has lower performance and therefore lower area than the proposed MMCM HLS implementation. In addition, it requires higher clock frequency to achieve real time performance. The handwritten HEVC fractional interpolation hardware implementation proposed in [18] has both lower performance and larger area than the proposed MMCM HLS implementation. The handwritten HEVC fractional interpolation hardware implementation proposed in [27] has higher performance than the proposed MMCM HLS implementation at the expense of larger area.
3.2 FPGA Implementations of HEVC Fractional Motion Estimation Using High-Level Synthesis

3.2.1 HEVC Fractional Motion Estimation Algorithm

After integer pixel motion estimation is performed for a PU, FME is performed for the same PU to obtain fractional-pixel accurate motion vector (MV). In HEVC reference software video encoder (HM) [5], FME is performed in two stages. As shown in Figure 3.4, 8 sub-pixel search locations around the best integer pixel search location are searched in the first stage. 8 sub-pixel search locations around the best sub-pixel search location of the first stage are searched in the second stage.

HEVC FME first interpolates the necessary sub-pixels for sub-pixel search locations using three different 8-tap FIR filters. In Figure 3.4, half-pixels a, b, c and d, h, n are interpolated using the nearest integer pixels in horizontal and vertical directions, respectively. Quarter-pixels e, i, p and f, j, q and g, k, r are interpolated using the nearest a, b and c half-pixels, respectively. HEVC FME then calculates the SAD values, as shown in (3.4) for each sub-pixel search location, and determines the best sub-pixel search location with the minimum SAD value.

\[
SAD = \sum_{i=0}^{m} \sum_{j=0}^{n} |R_{ij} - C_{ij}| \quad (3.4)
\]

\[
m = 0 \text{ to } (PU_{width} - 1), \quad n = 0 \text{ to } (PU_{height} - 1)
\]

![Figure 3.4 Sub-pixel Search Locations](image.png)
HEVC performs fractional motion estimation for 24 different PU sizes (4x8, 8x4, 8x8, 4x16, 16x4, 8x16, 16x8, 12x16, 16x12, 16x16, 8x32, 32x8, 16x32, 32x16, 24x32, 32x24, 32x32, 16x64, 64x16, 32x64, 64x32, 48x64, 64x48 and 64x64). There are 593 different PUs for these 24 different PU sizes, and 593 different SAD values should be calculated for them.

3.2.2 FPGA Implementations

The proposed HEVC fractional motion estimation HLS implementation for 8x8 PU size is shown in Figure 3.5. Three HEVC fractional motion estimation HLS implementations are done. In the first one (MM), in the C codes, multiplications with constants are implemented using multiplication operations. In the second one (MAS), multiplications with constants are implemented using addition and shift operations. In the last one (MMCM), addition and shift operations are implemented using Hcub multiplierless constant multiplication algorithm [22].

Fractional interpolation is implemented as described in Section 3.1. However, in the proposed FME HLS implementation, 16 integer pixels are taken as input instead of 15 integer pixels for all the necessary SAD calculations. There are 3 9x8 memories for d, h and n half pixels, 3 16x9 memories for a, b, and c half pixels, and 9 9x9 memories for quarter pixels.

In the first stage, 8 fractional pixel search locations around the best integer pixel search location are searched. 8 parallel SAD calculation hardware are used to calculate SAD values of these 8 search locations in parallel. Appropriate current, half and quarter pixels are read from current, half and quarter pixel memories, respectively, for the SAD calculations. 8 parallel absolute difference (AD) hardware calculate AD values of an 8x8 PU in 8 clock cycles. Then, SAD value of this 8x8 PU is calculated using these ADs. After the SAD values are calculated, comparison hardware determines the search location with minimum SAD value.

In the second stage, 8 fractional pixel search locations around the best fractional pixel search location of the first stage are searched. The same hardware used in the first stage is used for SAD calculation. After the SAD values are calculated, comparison hardware determines the search location with minimum SAD value.

Finally, the minimum SAD value found in the FME is compared with the SAD value of the best integer pixel search location, and the search location with minimum SAD value is determined.
The proposed MM and MAS FME HLS implementations for 8x8 PU size are extended to support almost all (22 out of 24) PU sizes (8x8, 16x8, 8x16, 16x16, 32x8, 8x32, 32x16, 16x32, 32x24, 24x32, 32x32, 16x64, 64x16, 64x32, 32x64, 64x48, and 64x64). For PU sizes larger than 8x8, PUs can be divided into 8x8 pixel blocks. Therefore, the proposed FME HLS implementation for 8x8 PU size is parameterized to support larger PU sizes. All loops are parameterized to satisfy the number of iterations necessary for specific PU size. Because of the asymmetric PU sizes, all loops are designed as nested loops. Also, memory sizes are arranged to support different PU sizes.

ALC directive is used for subtraction, addition, multiplication, and shifting operations to decrease hardware area. Pipeline (PIPE) directive is used between functions, for loop iterations, and computations. PIPE decreases latency and increases frequency of proposed FME HLS implementations. Resource (RES) directive is only used for memories. Some arrays are forced to map to BRAM instead of registers using
RES directive to decrease hardware area. AMAP directive is used to store half pixels in the memory efficiently. APAR directive is used to use registers instead of BRAMs. This increases hardware area. Since APAR provides parallel data accesses, it increases the performance. In addition, bit accurate model is used to decrease adder bit widths and therefore hardware area.

FPGA implementation results for the HLS implementations of HEVC fractional motion estimation algorithm are shown in Table 3.6. As shown in Table 3.6, the proposed FME HLS implementation results are divided into two groups; (i) for only 8x8 PUs, and (ii) for all PU sizes. There are three different FME HLS implementations (MM, MAS and MMCM) for only 8x8 PUs. Allocation (ALC), pipeline (PIPE), resource (RES) and array partition (APAR) directives are used in these HLS implementations. There are two different FME HLS implementations (MM, MAS) for all PU sizes. The best results for these HLS implementations are shown in Table 3.6.

As shown in Table 3.6, allocation and pipeline directives directly affect the performance of the proposed HLS implementations. Allocation limits number of resources used. Therefore, ALC directive decreases the number of DSP48 units for multiplication operations, and LUTs for the addition/subtraction operations. Pipeline directive decreases the number of clock cycles and increases the performance of the proposed HLS implementations.

The effect of the allocation directive for MM HLS implementation is analyzed in Table 3.7. Number of DSP48 blocks, clock cycles and frequency are observed by changing the number of available multipliers. Increasing the number of multipliers after a threshold value do not affect the results because of the data dependencies. Decreasing the number of multipliers increases the complexity of control because of the complex resource sharing mechanism. This reduces the frequency.
Table 3.6 HEVC Fractional Motion Estimation HLS Implementation Results

<table>
<thead>
<tr>
<th>PU Design</th>
<th>Optimizations</th>
<th>Slice</th>
<th>LUT</th>
<th>DFF</th>
<th>BRAM</th>
<th>DSP48</th>
<th>Freq.</th>
<th>Clock Cycles</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>NOOPT</td>
<td>8743</td>
<td>24722</td>
<td>10309</td>
<td>9</td>
<td>202</td>
<td>72</td>
<td>1304</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>ALC(M500)</td>
<td>10088</td>
<td>29707</td>
<td>21741</td>
<td>9</td>
<td>38</td>
<td>125</td>
<td>1501</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>ALC(M500)_APAR_PIPE_RES(BRAM)_BIT</td>
<td>12767</td>
<td>36761</td>
<td>26875</td>
<td>44</td>
<td>146</td>
<td>125</td>
<td>201</td>
<td>19.2</td>
</tr>
<tr>
<td>8x8</td>
<td>NOOPT</td>
<td>11800</td>
<td>33805</td>
<td>24458</td>
<td>9</td>
<td>0</td>
<td>143</td>
<td>1501</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>ALC(A20_S20)_PIPE</td>
<td>11077</td>
<td>32424</td>
<td>27486</td>
<td>10</td>
<td>0</td>
<td>143</td>
<td>1219</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>ALC(A20_S20)_APAR_PIPE_BIT</td>
<td>17155</td>
<td>52449</td>
<td>42093</td>
<td>41</td>
<td>0</td>
<td>125</td>
<td>241</td>
<td>16</td>
</tr>
<tr>
<td>MM</td>
<td>NOOPT</td>
<td>10226</td>
<td>29196</td>
<td>22889</td>
<td>6</td>
<td>0</td>
<td>167</td>
<td>713</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>PIPE</td>
<td>9735</td>
<td>28458</td>
<td>21922</td>
<td>6</td>
<td>0</td>
<td>143</td>
<td>453</td>
<td>9.7</td>
</tr>
<tr>
<td></td>
<td>ALC(A100_S100)_APAR_PIPE_BIT</td>
<td>16366</td>
<td>52521</td>
<td>41535</td>
<td>0</td>
<td>0</td>
<td>167</td>
<td>140</td>
<td>36.8</td>
</tr>
<tr>
<td>MM</td>
<td>ALC(M20)_APAR_PIPE_RES(BRAM)_BIT</td>
<td>13027</td>
<td>41397</td>
<td>21864</td>
<td>69</td>
<td>57</td>
<td>111</td>
<td>9024</td>
<td>24.3</td>
</tr>
<tr>
<td></td>
<td>ALC(A20_S20)_APAR_PIPE_BIT</td>
<td>13632</td>
<td>41085</td>
<td>22545</td>
<td>69</td>
<td>10</td>
<td>143</td>
<td>9051</td>
<td>31.2</td>
</tr>
</tbody>
</table>

Table 3.7 Allocation Analysis for MM HLS Implementations

<table>
<thead>
<tr>
<th>Fract. Interp.</th>
<th>M1</th>
<th>M10</th>
<th>M50</th>
<th>M100</th>
<th>M200</th>
<th>M500</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP48</td>
<td>32</td>
<td>58</td>
<td>104</td>
<td>135</td>
<td>135</td>
<td>---</td>
</tr>
<tr>
<td>C. Cyc.</td>
<td>1133</td>
<td>196</td>
<td>156</td>
<td>148</td>
<td>148</td>
<td>---</td>
</tr>
<tr>
<td>Freq.</td>
<td>165</td>
<td>167</td>
<td>170</td>
<td>170</td>
<td>170</td>
<td>---</td>
</tr>
<tr>
<td>FME (8x8)</td>
<td>DSP48</td>
<td>0</td>
<td>2</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>C. Cyc.</td>
<td>1901</td>
<td>1501</td>
<td>1501</td>
<td>1501</td>
<td>1501</td>
</tr>
<tr>
<td></td>
<td>Freq.</td>
<td>125</td>
<td>130</td>
<td>130</td>
<td>129</td>
<td>129</td>
</tr>
</tbody>
</table>

The proposed HEVC FME HLS implementation is compared with the handwritten HEVC FME hardware implementations in the literature [30] - [33]. As shown in Table 3.8, [30] has smaller area and higher performance than the proposed hardware. However, it interpolates SADs instead of pixels. Therefore, it decreases PSNR and increases bit rate. In [31], FME hardware searches all possible 48 sub-pixel search locations. However, it only supports square shaped PU sizes. In [32], FME
hardware supports all PU sizes but 8x4, 4x8 and 8x8. It uses bilinear filter for quarter-pixel interpolation. Also, it searches 12 sub-pixel search locations. In [33], FME hardware supports all PU sizes but it uses a scalable search pattern.

Table 3.8 HEVC Fractional Motion Estimation Hardware Comparison

<table>
<thead>
<tr>
<th>Tech.</th>
<th>Gate/Slice Count</th>
<th>Freq. (MHz)</th>
<th>PU Sizes</th>
<th>Fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[30] Xilinx Virtex 6</td>
<td>1814</td>
<td>142</td>
<td>All</td>
<td>19 QFHD</td>
</tr>
<tr>
<td>[31] 65 nm</td>
<td>249.1 K</td>
<td>396</td>
<td>Square Shaped</td>
<td>6 QFHD</td>
</tr>
<tr>
<td>[32] 65 nm</td>
<td>1183 K</td>
<td>188</td>
<td>All but 8x8, 8x4, 4x8</td>
<td>15 QFHD</td>
</tr>
<tr>
<td>[33] Xilinx Virtex 6</td>
<td>130 K</td>
<td>200</td>
<td>All</td>
<td>32 QFHD</td>
</tr>
<tr>
<td>Prop. Xilinx Virtex 6</td>
<td>13632</td>
<td>143</td>
<td>All but 4x8, 8x4</td>
<td>8 QFHD</td>
</tr>
</tbody>
</table>
CHAPTER IV

CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed the first FPGA implementation of HEVC full search motion estimation using Vivado HLS. Then, we proposed the first FPGA implementations of two fast search (diamond search and TZ search) algorithms using Vivado HLS. Finally, we proposed the first FPGA implementations of HEVC fractional interpolation and motion estimation using Vivado HLS. All FPGA implementations are verified to work correctly at real-time using post place and route simulations.

As future work, FPGA implementations of fast search motion estimation algorithms can be extended for variable block sizes and variable search ranges. FPGA implementations of other fast search algorithms such as hexagon search can be done using Vivado HLS.
BIBLIOGRAPHY


